COMP1130 Assignment 2

Manindra de Mel, u7156805

Tutor: James, LAB: Tuesday 13:00 - 15:00

Introduction

Assignment two is an automaton which is similar to Conway's game of life. The assignment provides all the graphical functionality required to run the program. However, the rules surrounding the interaction between cells and the calculations of each generation are all functionalities that need to be implemented. Additionally, an extension task specifying a new set of rules was also made as a direct comparison to Conway's game, showing how different an automaton can be with a different set of rules.

Program functionality

Once opened, the user can switch between the two automaton or try the initial `QRworld` automaton, which can also be switched to with the `Q` key press. In `QRworld` the user can switch between one of three sample images with the keys `1 - 3`. Additionally, the user can click on an individual cell to rotate between the different states provided by `QRworld`. In terms of different generations, the user can use the `+/-` keys to increase or decrease the step of iterations and then use the `spacebar` to jump with the specified iterations. Similarly, `.` is used to only jump 1 iteration. This same functionality is also present in the extension automaton `Battle world` which can be switched to with the `b` key press.

QR World

In `QRworld` there are two types of cells present within an arbitrary grid. A cell can either be `dead` or `alive` and is called the cell's state. Then on each iteration, the program must go through each cell and determine its next state based on the set of rules. The cell must first determine its neighbours *(not including diagonals)*. Given the cell is `alive` there must be at least two but no more than 3 `alive` neighbours to remain `alive`. On the contrary, if a cell is `dead` there must be either 2 or 4 `alive` neighbours to 'reproduce' and switch the state to `alive`.

Helper functions

The problem can be broken down to a series of helper functions, all of which build up to `nextGenQR`. We first need to get the neighbours of the cell we're currently on. The first concern with this function are the bounds of the grid. If we're checking the neighbours of a corner cell, then it will only have 2 neighbours. On the contrary, unbounded cells will typically have 4 neighbours. As such we can create a function `getNeighboursCoords :: Grid c -> GridCoord -> [GridCoord]` which will take the grid, the point we're focusing on and return the surrounding neighbour coordinates in a list. Once we have the coordinates, we have to turn both the cell point and it's neighbours into it's associated states on the grid. This can be done with our `getForNextGen :: Grid c -> GridCoord -> c` which converts a `GridCoord` into its respective state. At this point of our program, I made the decision to store the cell point and it's neighbours in a hash map fashion, where we have `[(Alive, [Dead, Alive, ...])]`. This structure allows easy differentiation between different cells and it's neighbours. Finally, we can then pass this list of cells and it's neighbours to the function which determines the next state of that cell, known as `nextState`.

Main functions

With our helper functions completed, we now have to piece them together in our main function `nextGenQR`. Again, this can be done in a successive fashion. First we generate all the coordinates with `allCoords` and then get the neighbours of each coordinate by mapping our `getNeighBoursCoords` to every point. We now have a list that looks like this `[[(0,1), (1,0)], [(0,0), (1,1), (2,0)] ..]`. However, we now want to convert this to states by

using our `getForNextGen` function to get something like `[[Alive, Dead], [Alive, Dead, Alive]..]` and we can encapsulate this with list comprehension.

```
neighbourPointsList = [map (getForNextGen (Grid a b c)) points | points <- map
(getNeighboursCoords (Grid a b c)
(allCoords a b)]
```

We then need to pair each neighbour list with its original point. So similar to the neighbours list we create all the points with `allCoords` then convert it to it's state with `getForNextGen`. Finally, we can then get our new grid by using `zipWith` our original point list and our neighbour list with the function `nextState`.

```
newC = zipWith (curry nextState) (map (getForNextGen (Grid a b c)) (allCoords
a b)) neighbourPointsList
```

As shown above. All the steps taken, from creating and defining helper functions to piecing them together in the main `nextGenQR` function, were all taken in a logical method. This means that for anyone reading the code, it should be easy to understand how each function is being utilised, and it's purpose.

Battle World

Battle world is the extension task that I created as another cellular automaton for the user to interact with. Although the user interaction with `BattleWorld` is exactly the same as `QRworld`, the internal rules of the automaton are both vastly different and arguably more complex.

As the name suggests, Battle world consists of a 'Battle', in my automaton, there are two teams with the goal of eliminating as much of the enemy as possible. Team 1 is displayed as red, whilst Team 2 is displayed as Blue. The light green are the `Ground` states which the two teams move on. Currently, there are 2 phases to each iteration of this automaton; The movement phase and the attacking phase. The movement phase is what makes `BattleWorld` fundamentally different to `QRworld`. In the movement phase, each cell on a given team must move to the closest enemy cell. Then in the attack phase, there are a set of rules regarding the neighbours, similar to `QRworld`'s `nextState` function. In this case, we check for the amount of allies and enemies around a given cell *(including diagonal's)*. We first check that if there's at least 1 enemy, and there aren't enough allies to support the cell, it will die and become a `Ground` state. This applies to both Team 1 and Team 2.

Helper functions

Moving the soldiers

Battle world required a series of new helper functions to create each generation of the automaton. First, we have to begin with the movement phase. For each cell we need to get the closest enemy cell *(if it exists)* which was done with `getClosestSpecificObject`. Once we have the enemy point, we need to discern which adjacent cell to move to, in order to move towards it. This task was achieved by `getNextPoint`. We then need to clarify that the point the cell is moving to on the grid is a valid point; the point must be a ground point. This was done to avoid ally points from randomly disappearing (`validMovement`). Finally, we can use all of these functions and piece them together in `nextMovementPhase`. `nextMovementPhase` is an important function which takes the grid and `[GridCoord]` and returns another grid with all the moved soldiers. The function does this recursively, by taking the head of the

coordinate list and then moving it, respectively. It applies all the functions we mentioned before, it gets the closest enemy, gets the next point and checks if the point is valid. However, there is also added functionality, given the point is valid. The function sets the new point to the current cell's state and sets the current cell to `Ground` to avoid duplications. The `moveSoldiers` function then applies `nextMovementPhase` to a Grid and it's coordinates.

Attacking with soldiers

The attacking phase was implemented in an almost identical manner to `QRworld`. We first need to modify the `getNeighboursCoords` function to include diagonal neighbours. Then, like `QRworld` a neighbour list was generated and then paired with its original cell point `(BattleCell, [BattleCell, BattleCell, ..])`, all of which were passed to `nextState` to decide the next state of the given cell.

Main functions

In `BattleWorld`, we have two main functions `moveSoldiers` and `nextGenBattle`, which represent each phase of the battle, respectively. As explained in the helper functions. `moveSoldiers` moves each team cell to the enemy. Whilst `nextGenBattle`, similar to `nextGenQR` gets the next state of each cell with respect to its neighbours and either kills the cell or keeps it alive depending on the number of allies and enemies around it.

Testing

Testing in the helper functions

Several testing forms were used throughout the assignment, most of the used testing practices were due to the Week 7 lab. Initially, most of the testing for the program was black box testing found in `AutomataTest.hs`. However, as I changed my type definitions and how functions interacted with each other, I resolved to more white box testing. In total, I had `27` tests for all the various functions in both `QRworld` and `BattleWorld`. These tests often tested more of the complex functionality, which the program required to work bug-free and in precision. For example, a black box test was written for `getNextPoint` and it was soon prevalent that the output was differing to that of the expected output. This is because team cells were suddenly being overwritten due to previous cells overwriting them on the movement phase, this then led to the creation of `validMovement` which resolved this issue.

Testing in the main functions

The main functions could not be tested in a manner that similar to that of the helper functions. This is because of the nature of its return type, being unable to be compared. I could have resolved this by letting the datatype derive Eq. However, I found my method of testing to be more reliable, and it is testing the same equivalence. Some of the these functions include `nextGenQR`, `moveSoldiers` and `nextMovementPhase`. With these functions I made a small test Grid, often a 3×3 or 2×2 grid and then calculate the next generation by hand on paper, physically drawing out the current and next grid. I would then run those specific functions again and display it on the web-based interface to confirm the function is working properly.

Reflection

Overall, the functionality present within the program was satisfying to both implement and use. However, this being said, there are some aspects of the program which I wish were improved upon. On the minor side of things, one assumption I made for the user is that they already knew about all the key binds defined in `App.hs`, as there is no graphical instruction on

how to use the program. Similarly, my program is not as efficient as I think it could be. Given more time, I believe I could've reduced both the time and space inefficiencies present within my program. Noticeably, in `BattleWorld` instead of having two passes over the grid on each iteration, maybe there was a method to have only one pass over the Grid. However, the solution would most likely be convoluted and the logical establishment in my program would be lost for the benefit of efficiency.

Finally, the major design change I made whilst creating my extension task was its complexity. Initially I was going to have the `BattleWorld` datatype as defined below:

```
type Range = Int
type inBush = Bool
data BattleCell = Team1 Range InBush | Team2 Range InBush | Bush | Ground
```

As shown, there was to be a lot more complexity, involving mechanics like ranged combat and cover which reduces the range of enemies firing *(ranged weapons)* at the cell. However, due to the technical complexity in implementing more graphical based user inputs and involving range proved the task to be too complex given the amount of time. Therefore, given more time, I would definitely have a harder attempt at implementing a more complex automaton with more complex rules to show a more interesting behaviour.